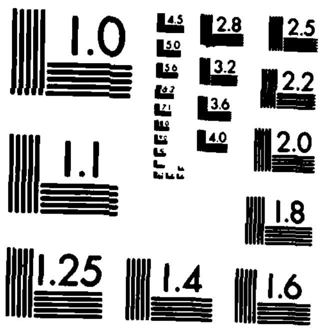
AD-A146 575   REUSABLE SOFTWARE(U) NAVAL POSTGRADUATE SCHOOL MONTEREY   1/1
CA W C JOHNSON MAR 84

UNCLASSIFIED   F/G 9/2   NL



˜OPY RESOLUTION TEST CHART

2

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

REUSABLE SOFTWARE

by

William C. Johnson

March 1984

Thesis Advisors: G. H. Bradley
N. R. Lyons

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. AD-A146 572	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Reusable Software		5. TYPE OF REPORT & PERIOD COVERED Master's Thesis; March 1984
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) William C. Johnson		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, California 93943		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, California 93943		12. REPORT DATE March 1984
		13. NUMBER OF PAGES 68
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Reusable Software

Reusable Design

Design With Existing Components

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This thesis reviews the topic of software reusability with special emphasis upon the reusability of products of the design phase of the software life cycle. The ideas of software reuse as a capital-intensive process and reuse of products of all phases of the software life cycle are also presented. The thesis presents a formal definition of the term software reusability, presents a hypothetical design scenario incorporating reuse,

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

S/N 0102-LF-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

#20 - ABSTRACT - (CONTINUED)

and compares the requirements of a reusable software design methodology with features of existing design methodologies. Other issues pertinent to software reuse in general, and reuse of design in particular, are reviewed.

Reusable Software

by

William C. Johnson
Lieutenant Commander, Medical Service Corps, United States Navy
B.A., University of the South, 1973
M.S., Trinity University, 1976

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN INFORMATION SYSTEMS

from the

NAVAL POSTGRADUATE SCHOOL
March 1984

Author: William C. Johnson

Approved by: [illegible]
Thesis Co-Advisor

Norman R Lyons
Thesis Co-Advisor

Richard S. Elster
Chairman, Department of Administrative Sciences

[illegible]
Dean of Information and Policy Sciences

ABSTRACT

This thesis reviews the topic of software reusability with special emphasis upon the reusability of products of the design phase of the software life cycle. The ideas of software reuse as a capital-intensive process and reuse of products of all phases of the software life cycle are also presented. The thesis presents a formal definition of the term software reusability, presents a hypothetical design scenario incorporating reuse, and compares the requirements of a reusable software design methodology with features of existing design methodologies. Other issues pertinent to software reuse in general, and reuse of design in particular, are reviewed.

TABLE OF CONTENTS

I. INTRODUCTION AND BACKGROUND ---------------------- 7

A. THE SOFTWARE CRISIS -------------------------- 7

B. OVERVIEW OF SOFTWARE REUSABILITY ------------- 12

C. OUTLINE OF THESIS ---------------------------- 17

II. DEFINITION OF REUSABLE SOFTWARE ----------------- 19

A. SOFTWARE AS CAPITAL -------------------------- 19

B. LIFE CYCLE IMPLICATIONS ---------------------- 23

C. DEFINITION ------------------------------------ 26

III. OVERVIEW OF SOFTWARE DESIGN --------------------- 27

A. DESCRIPTION OF SOFTWARE DESIGN --------------- 27

B. CURRENT DESIGN METHODOLOGIES ----------------- 28

IV. REUSABLE SOFTWARE DESIGN LITERATURE -------------- 35

A. RAYTHEON EXPERIENCE -------------------------- 35

B. ASSORTED REFERENCES -------------------------- 37

C. JLC PANEL ON SOFTWARE REUSABILITY ------------ 40

D. ITT WORKSHOP ON REUSABILITY ------------------ 41

E. SUMMARY AND ANALYSIS OF SOFTWARE DESIGN LITERATURE ------------------------------------ 43

V. GUIDELINES FOR A REUSABLE DESIGN METHODOLOGY ----- 46

A. A PROPOSED REUSABLE DESIGN SCENARIO ---------- 46

B. REUSABLE DESIGN METHODOLOGY CHARACTERISTICS -- 49

C. CURRENT DESIGN METHODOLOGIES COMPARISON ------ 50

D. AN IDEALIZED REUSABLE DESIGN METHODOLOGY ----- 53

VI. OTHER ISSUES -- 55

A. PROJECT ORGANIZATION AND CONTROL ------------- 55

B. QUALITY ASSURANCE/TESTING -------------------- 56

C. REUSE ARCHIVAL ISSUES ------------------------ 58

VII. CONCLUSIONS AND RECOMMENDATIONS ------------------ 61

LIST OF REFERENCES ----------------------------------- 64

INITIAL DISTRIBUTION LIST ----------------------------- 68

I. INTRODUCTION AND BACKGROUND

A. THE SOFTWARE CRISIS

The motivation behind research into reusable software lies in the current software crisis and in reusable software's possible role in alleviating that crisis. The term software crisis denotes a situation within the computer industry in which production and maintenance of computer systems is "bottlenecked" by the software components of these systems. This represents a reversal of the situation in the late 1940's and 1950's when difficulties in system development were primarily caused by hardware. Indeed, much attention in software development during this period was given to the optimization of scarce hardware resources.

Since the beginning of the computer industry in the late 1940's, hardware production has been marked by two trends which have reduced the hardware bottleneck and resulted in the current software crisis. The first of these is the advancement of hardware technology; that is, from vacuum tubes to transistors to integrated circuits. The result is the elimination of hardware as a scarce resource in many systems. Second and more important, the application of techniques of mass production, borrowed from other industries, is changing the production of computer hardware from a custom craft to a true assembly line process where multiple, identical units are produced over extended production runs. The hardware side of the computer industry is following the pattern of automobile, household appliance and other industries.

These same trends are not present in production and maintenance of computer software. Indeed, software design is viewed as being analogous to production of artwork or an exspensive automobile. It is a task in which components are individually developed by highly skilled artisans and meticulously fitted together. Pieces which later on prove to be faulty or must be altered to meet new environmental constraints must be customized or replaced by a similar process. Despite piecemeal attempts to automate various stages in software production, the entire process remains an extremely labor intensive endeavor which starkly contrasts to the capital intensive nature of hardware production.

Recognition of existence of a crisis in software production and maintenance is generally thought to have occurred in the late 1960's, beginning with the NATO Software Conference in 1968. At this time, the industry first began to perceive that the real impediment to production of computer systems lay not in hardware but in software. Since this conference, attention is now focused on methods to eliminate the software bottleneck. An examination of some more important ideas, developed during this period which may have a bearing on resuable software, is now in order.

1. Structured Programming

Edsger Dijkstra is generally credited with popularizing the concept of structured or "go-to-less" programming. This began in 1968 with publication of his letter to the editor of the Communications of the ACM [Ref. 1] and has become a widely accepted technique of good programming. Its essence is summarized as being replacement of random, haphazard program structure with purposeful, cogent, nonrandom structure. This structure, it is asserted, is easier to understand, thus facilitating development and maintenance. Structured programming is associated with third and fourth generation block-structured languages which emphasize nested blocks of code.

2. Modularization

Modularization is an important development which is closely linked with structured programming. Modularity involves decomposition of complex programs into smaller segments or modules; each module performs a relatively simple and independent function. In this manner software becomes more manageable, more flexible and more comprehensible [Ref. 2].

3. Information Hiding

David Parnas is considered to be the originator of the concept of information hiding [Ref. 3]. This is the principle that the implementor of a module should possess only information needed to implement that module and nothing more and that the user of a module should possess only information needed to use that module and nothing more.

This thesis views structured programming, modularity and information hiding as concepts which can assist in efficient production of software; they make the production of software analogous to that of stereo equipment in which components are modularized and can be easily interchanged. Much of the literature on reusability recognizes that the concept of interchangeability should apply to software production; more efficient software production should be a consequence.

4. Software Life Cycle

An important occurrence since 1968 is the recognition that the production of software is a multi-stage process and not a single event in time. Rather than viewing production of software as simply production of program code, software engineers now emphasize that most software effort and costs occur during tasks other than coding. This view of software production as a time-sequenced, multi-staged process is known as the Software Life Cycle, of which there are many versions. For this thesis, Barry Boehm's Waterfall Software Life Cycle Model is used and appears as Figure 1. It is perceived that overall solutions to the software crisis can not focus exclusively on any single stage of the software production cycle; instead, all stages must be considered.

Overall, the history of the software crisis is still in the process of evolution with no final solution to the

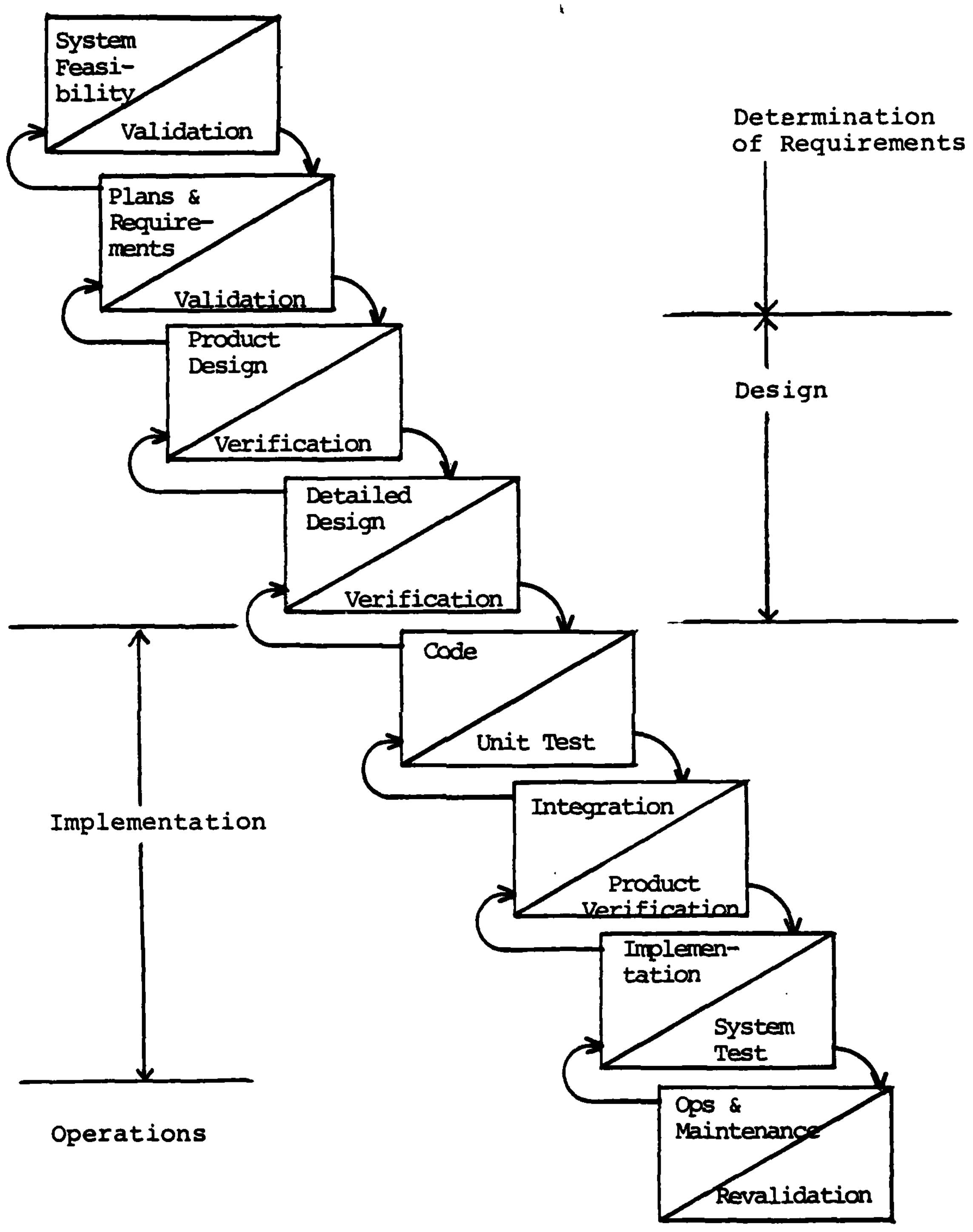


Figure 1. Waterfall Software Lifecycle Model

effective production of software yet having been found. Each of the ideas mentioned above, as well as other developments not mentioned, represents a partial solution. This thesis believes the software crisis results from an inability to convert a labor intensive production process into a capital intensive process; that is, to introduce automation and other techniques that utilize capital products, such as existing software components, into software development. Key to this conversion is a concept which has only recently been recognized as having potential to alleviate the software crisis. This is the concept of reusable software. A more precise definition of the concept awaits Chapter II of this thesis, but for now it is simply defined as reuse of software components within single programs and among multiple programs. Just as early industrialists saw that the key to mass production of tangible goods was development of standardized components which could be used interchangeably, so is reuse of standardized, interchangeable software components the key to imrpoved production of software products.

B. OVERVIEW OF SOFTWARE REUSABILITY

As a concept and as an implemented practice, reusable software has a very short history. For convenience, this history is divided into three periods. First is the period from 1968 until 1978. The oldest known reference to reusability as a concept, in and of itself, occurred in a paper by Doug McIlroy at the 1968 NATO Conference on Software

Engineering [Ref. 4]. At this conference, McIlroy advocated a "component manufacturing facility" utilizing a parameterized family of routines. Horowitz and Munson mention that his work was criticized for various reasons [Ref. 5], and the subject of software reuse seems to have disappeared for the next ten years. Of the articles either about software reuse or containing references to reuse, one finds none before 1978, other than McIlroy's. The second period occurs from 1978 to 1981 and is termed the "buzzword phase" by this thesis. During this period references to reusability occurred in the computer literature; however, they primarily cited the concept, usually as a possible solution to the software crisis. Most of the literature on reusability in this period is devoid of significant analysis of the concept, exceptions being those articles written by Robert Lanergan [Refs. 6-8]. Reusability during this period is, therefore, merely a "buzzword" which most authors mention without seriously analyzing. Of approximately 20 individual articles written on the subject, well over 60% have been written since 1981, which is the final period. This period is characterized by increased emphasis and analysis of reusability as a potential solution to the "software bottleneck." Research into reusability has increased and has been more in depth since 1981; indeed, two conferences have been held since then, one partially devoted to the subject, the other totally devoted to reuse of software.

1. Software Reusability Literature

The literature on software reusability consists of approximately 20 individual articles and the proceedings of two conferences. Perhaps most important of the articles are a series done by Robert Lanergan and others of Raytheon, Inc., describing that corporation's actual implementation of a reusable code methodology in software application development [Refs. 6-8]. Two articles by Rauch-Hindin explore possible gains from implementing reusable software as well as implementation obstacles [Refs. 9-10]. In two articles, Peter Wegner is the first to recognize software production as a capital intensive process and emphasize reusability's role in that process [Refs. 11-12]. In the IEEE Tutorial on Software Design Techniques, Peter Freeman published one of the first in-depth, analytical discussions of the concept [Ref. 13]. Most of the remaining literature on reusability, except that below, is characterized as being isolated, superficial overviews of, or references to, the subject.

2. NPGS Software Workshop

Due to its limited distribution, the Proceedings of the Software Workshop [Ref. 14] held at NPGS, Monterey, Ca., is not included in the literature cited above. At this workshop, one of the panels was devoted exclusively to the topic of software reusability. Although oriented toward a DOD perspective, most of the panel's analysis applies to reusability in general and represents the first attempt to analyze the subject comprehensively. Especially important is the panel's consideration of life cycle implications for reuse of software; prior to this report, the literature seems to focus on reuse of code alone. In addition, the Workshop attempts to deal with some of the "how to" questions of reuse implementations, which were previously ignored.

3. <u>ITT Workshop on Reusability</u>

Most recently, ITT sponsored a workshop solely devoted to reusability and subsequently published the proceedings. Unlike the NPGS workshop which represents a unified approach, the ITT <u>Proceedings</u> [Ref. 15] is a collection of articles on a variety of reusability topics. As such, a multiplicity of concepts and viewpoints are presented with no single focus. However, the diversity of ideas is an asset.

4. <u>Software Reuse Implementations</u>

Perhaps the one aspect of software reusability which most reveals its immaturity is the number of actual implementations. As mentioned above, Raytheon, Inc., appears to be the first and most extensive implementation of reusability. This is a data processing implementation using COBOL as source language. Other implementations include Hartford Insurance Company, Security Pacific Bank in Los Angeles, Houghton Mifflin and Xerox[2].

The literature on software reuse cites several different computer applications as examples of existing

[2]Personal telephone conversation with Robert Lanergan.

forms of reuse. For example, Horowitz and Munson list the following:

1. subroutine libraries
    a. general libraries--utilities, etc.
    b. specific application areas--SPSS, etc.
2. compilers
3. simulation--GPSS, etc.
4. parameterized systems--automated program development based on user supplied information [Ref. 16].

Standish cites seven examples of existing forms of reuse:

1. mathematical subroutine libraries
2. operating system service calls
3. small granule capabilities--UNIX
4. large granule tools--EMACS
5. reuse of algorithms from books
6. software application generators
7. Japanese software factories [Ref. 17].

There are several conclusions to be drawn from the above lists. First, there is no agreement in the general literature on a standard definition of software reusability. As it will be derived in the next chapter, the definition of reusable software according to this thesis excludes several of the examples cited above. For example, our definition excludes subroutine libraries because their reuse requires no adaptation, nor are they reused in end uses different from that for which they were originally developed. This illustrates the need for a standard definition of the term. Second, those existing examples which can be classified as reuse by our definition are few and fairly recent; e.g., UNIX. This is to say that the field has not been innundated with experiments, much less been given serious consideration. Prior to 1978, the current concept of reusability, in and of itself, did not exist.

C. OUTLINE OF THESIS

Thus far this thesis has explored briefly the nature of the software crisis; it has been asserted that this arises from disparities between hardware and software production techniques. The former is more capital intensive, while the latter is more labor intensive. Finally, software reusability has been proposed as a key element in the "capitalization" of software production. As the brief overview of reusability shows, research on the specifics of implementing reusability is scant and fragmented at best.

This thesis seeks to add another building block to the structure which has been started by others. For reasons which will become clear in subsequent chapters, this task is best accomplished by an analysis focusing on the design stage of the software life cycle and its implications for reusability. However, this requires that adequate attention be given to the whole concept of software reuse. Two major subtasks are necessary. First, a synthesis of existing, important knowledge is required; this is attempted in this chapter and Chapter II, where software reusability is defined. This synthesis continues throughout the remaining chapters. The second major subtask is to develop a framework for understanding and implementing reusability. This is done by adoption of a design perspective in Chapters III, IV and V. Chapter III specifically overviews software design, Chapter IV reviews the literature on reuse of design, and Chapter V develops guidelines for a reusable design methodology. Chapter VI develops other issues pertinent to software reuse in general that also affect the design issue. Chapter VII contains conclusions and recommendations.

II. DEFINITION OF REUSABLE SOFTWARE

The purposes of this chapter are to fill a void regarding the exact meaning of the term "reusable software" and to understand its software life cycle implications. What is meant by the term? What types of software does it encompass? Is an operating system which is continually reused by application programs to be considered reusable software? Are off-the-shelf commercially purchased programs which are used in their entirety to be considered as reusable software? Until now, the majority of references to the term have not attempted to answer these kinds of questions, nor have they attempted to define the term itself. Most references tend to refer to the term as if it were an intrinsically worthy objective whose meaning is intuitively understood. Our purpose must be to bring more precision to the term, so that further research into characteristics and structure of reusable software may be based on a concrete understanding of the term.

A. SOFTWARE AS CAPITAL

An understanding of reusable software begins with knowledge that it is part of a process; the previous chapter alludes to this, but it must now be made more explicit. The process under consideration is the creation of software products, primarily the generation of programs to be run on computer hardware. This process is labor intensive, but attempts are being made to make it more capital intensive. It is today less labor intensive than it was thirty years ago, a trend which hopefully will continue. "A production process is capital-intensive if it involves expenditures early in its life cycle for the purpose of increasing productivity later in the life cycle" [Ref. 18]. Additionally, Wegner defined capital as a reusable resource, the cost of which can be distributed over the "set of actual or potential uses" of that resource [Ref. 19]. Thus, the heart of a capital intensive software production process is the existence of certain resources that can be utilized more than once after their initial development. Development costs for these resources should, in some way, be partitioned among users of this resource.

The problem of converting software production from a labor intensive into a capital intensive process actually involves two separate but related elements. First, it is a desirable, but not necessary, goal to automate the production process; this entails the substitution of computer effort for human effort into as many phases of the software life cycle process as possible, particularly in the latter stages of coding, integration, implementation and maintenance. Second, and more important, is that techniques that use existing components must be introduced into software development. What we are really seeking is not the ability to produce software by computers, although this is a worthy objective;

rather, we seek to produce software in greater quantities, just as automobiles or computer hardware are produced.

In classical economics, mass production is accomplished when an entrepreneur combines the three factors of production--land, labor and capital--via his organizational talents. The factory organization so constituted receives raw materials as inputs and processes them through the factory, and mass produced outputs result. In our modern economy, generally the entrepreneur is replaced by some form of corporate organization, but the basic process remains.

Although once developed, software can be replicated relatively simply, it is its development which remains labor intensive. The three factors of production in its development are primarily labor, combined with small amounts of land and capital. What is sought is a production oriented design process similar to the design processes involved in architecture, computer hardware development and other industries. In such a software design process, designs can be produced using standard components, standard design techniques and standard forms of communication. In such a system, capital becomes the dominant factor of production as software components, the raw materials of the system, become inputs to a "software factory." Application programs become the outputs. The unique thing about production of software under such a system is that software components, in addition to being raw materials, are also the capital of the organization since they are resources which can be utilized more than once after their initial development. Thus, reusable software components, along with computer hardware and any other equipment, comprise the capital of a software factory.

In equating software design with other forms of design, in this case computer hardware design, an interesting comparison occurs. Generally, the design process can be modelled as

"Design Idea" + "Standard Components" = "Finished Product"

Historically, hardware design is characterized by an increase in the size (i.e., number of units contained in a given component) of its standard components. Hardware design in the early 1950's, for example, is characterized by techniques where the largest standard component consisted of one or a few gates; now, it is a microchip containing hundreds of thousands of gates. Thus, hardware designers no longer have to work at the gate level; instead, they work at the register, or more likely, processor level. On the other hand, a corresponding increase in size has not occurred in software design. Even though software developers worked at the machine or assembly language level in the 1950's, they now work at the programming language statement level. To have maintained comparability with hardware designers, they should be designing with components comprised of thousands of programming language statements (or some elementary component of an earlier stage in the software life cycle). This level might be termed the module level. If this were the case, software design would become more capital intensive, and the capital component of software development would be improved.

B. LIFE CYCLE IMPLICATIONS

Having limited the term reusable software to mean reusable software resources of a capital nature, the concept must now be further refined; as it now stands, it is much too broad and all encompassing. Peter Freeman succintly puts it:

> We are not interested in the very important reuse of programs by multiple end-users upon multiple occasions (operating systems for example)...the focus is on the information needed by the developer (or maintainer). [Ref. 20]

It is important to distinguish the difference between mere reuse of an entire program without modification or adaptation and incorporation of a component or an entire program into another program, the product being a completely new program. We will define the latter to be an instance of reusable software, while the former is not. What is of interest is a function for which no previously known program exists and for which a completely new entity must be created, either by developing new programs or by combining/adapting previously existing structures.

A related problem which has frequently plagued uses of the term reusable software in the literature is the question of which reusable software resources of a capital nature are to be used in the development or maintenance of new software end uses. What sort of resources does this include? A common misconception seems to be that these resources only include program code itself and nothing else. One of the first groups to recognize the fallacy of this idea was the Panel on Software Reusability sponsored by the Joint Logistics Commanders. They defined reusable software to include "specification, design, code, and/or documentation" [Ref. 21]. Reusability must be viewed as encompassing the entire software life cycle, not just the implementation phase of the process. Indeed, Freeman stated that "reuse of program code alone has almost no value" [Ref. 22]. He proposes a hierarchy of types of information which can be reused; this is presented in Figure 2. The value of this diagram is its visual display of the wide array of software knowledge which may be reused. It conveys only an intuitive notion. The five levels of reusable information shown on the right hand side can be combined into three categories which more accurately portray major categories of reusable information:

1. external/environmental information--this category includes system documentation, requirements analyses, and other pre-design information;
2. logical structures/functional architectures--design information;
3. code fragments--program code, program documentation, test results, and other post-design information.

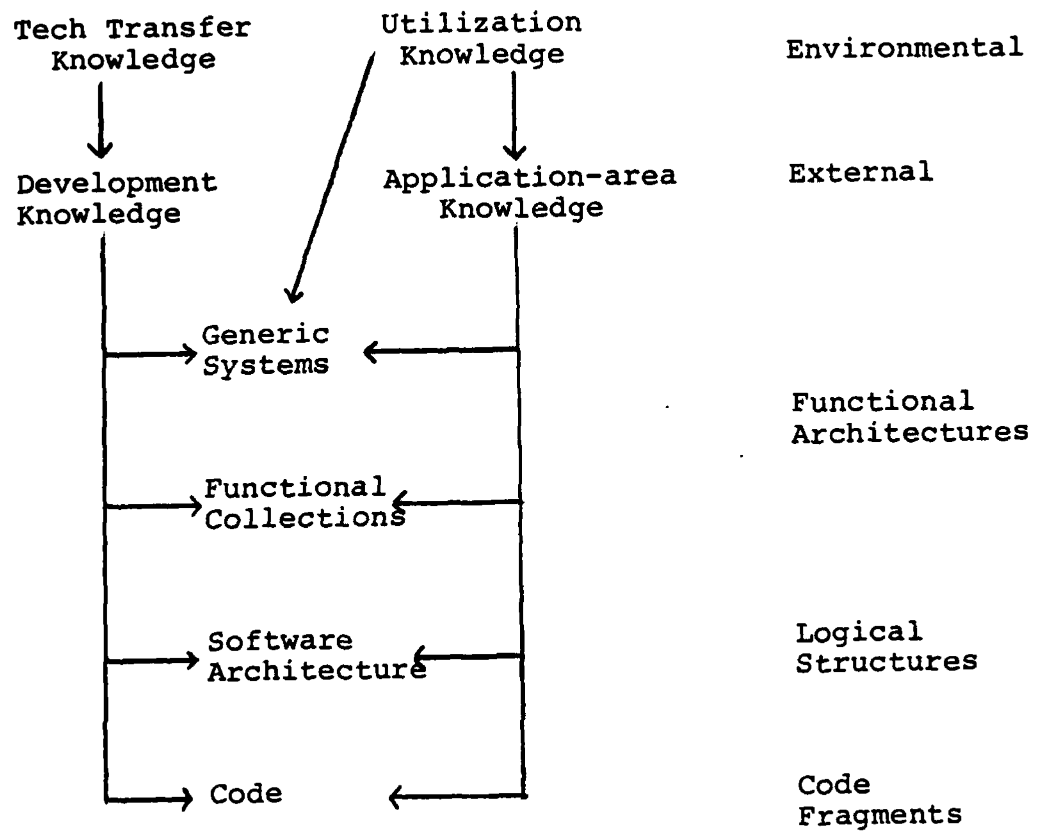


Figure 2. Hierarchy of Information

From Freeman's definitions of these levels in the hierarchy, it is evident that he intends reuse of software resources to be comprehensive. This thesis will also adopt this view.

This thesis utilizes the software life cycle framework as presented in Figure 1. Other frameworks are available; however, most reduce to a basic structure consisting of four primary stages: (1) determination of requirements, (2) design, (3) implementation and (4) operations. Differences are usually accounted for by the subdivisions of each of these four stages. As was mentioned in Chapter I, our emphasis is on the design stage, for this is perceived to be the crucial stage for implementing reusable software; however, in defining the term, all phases of the life cycle must be included.

C. DEFINITION

Reusable software is defined as software resources of a capital nature which are used in the development or maintenance of software products with end uses different from that of the component resources. These resources encompass any information generated at any time throughout the software life cycle. A component resource is described as a modular product of the software life cycle, possessing the characteristic of being highly cohesive.

III. OVERVIEW OF SOFTWARE DESIGN

The analysis of the design stage and its importance to reusable software begins by first describing the software design process. The second section of this chapter then reviews current design methodologies in order to gain a more detailed understanding of this process.

A. DESCRIPTION OF SOFTWARE DESIGN

While the requirements determination stage of the software life cycle specifies what is to be done, the design phase tells how to do it. The design process is an attempt "to achieve fitness between two entities: the form (problem solution) in question and its context (problem definition)" [Ref. 23]. Emphasis is on hierarchy, allocation of resources and satisficing [Ref. 24]. Freeman lists three purposes of design:

1. Discovery of problem structure.
2. Creation of solution outlines.
3. Review results in comparison with goals [Ref. 25].

Thus, the design process begins with some sort of problem stated as a requirement and produces a matching solution to that problem, expressed in one of many ways.

Most authors divide software design into two distinct phases. First is architectural or high level design where "emphasis is on determining the structure of the system, decomposing the system into modules, and precisely specifying the interfaces between the modules" [Ref. 26]. Second is detailed design in which algorithms are selected to implement modules developed in the architectural phase. The end result of software design is a representation of the system which can be used by the implementor.

B. CURRENT DESIGN METHODOLOGIES

Before discussing specific design techniques, there are five overall approaches to software design that one must consider. According to Freeman, these are:

1. Top-down--begin at the highest level of abstraction, then work down through successively lower levels.
2. Bottom-up--opposite of top-down in that lower level decisions are made first, emphasizing internal criteria.
3. Outside-in--essentially same as top-down with the sense of direction defined in terms of what the user sees.
4. Inside-out--opposite of outside-in since the implementation decisions are made before external (user-oriented) decisions.
5. Most-critical-component-first--most constrained or critical components are designed first [Ref. 27].

Of specific existing design techniques, six popular ones are examined. The first three of these may be classified as architectural design techniques, while the last three are detailed design techniques. Intent of this discussion is not to provide a cumulative, detailed discussion of design techniques. These six are chosen as a basis to show how the problem of implementing a reusable design methodology may be approached; they are also chosen because they represent differing approaches to design and thus offer an opportunity to examine reuse from many facets.

1. Structured Design

Structured Design is a technique normally associated with Constantine, Yourdon and Myers and is a graphical technique which utilizes a notation called a structure chart. Although similar in appearance to a traditional flowchart, a structure chart is based upon the concept of a module which refers "to a set of one or more contiguous statements having a name by which other parts of the system can invoke it" [Ref. 28]. Structure charts are developed using the following process:

1. Develop a functional picture of the problem.
2. Identify external conceptual streams of data. These are streams external to the software system and are independent of any physical input/output device.
3. Determine the points of highest abstraction from the major external conceptual stream. For an input steam, this point is where "data is fartherest removed from its physical input form yet can still be viewed as coming in." For an output stream, this occurs where the data is first viewed as going out.

4. Develop source and sink modules for each input and output stream at points of most abstract input and output data. Each module's function is described with a concise phrase, as well as any transformations that occur when that module is called.

5. Identify the last transformation necessary to produce the form being returned by each source module as well as the form of input prior to that transformation. For sink modules, identify the first process necessary to get closer to the desired output, and the resulting output form. Repeat on new source and sink modules until the original source and final sink modules are reached.

In general, the structure of the design should mesh with that of the problem, and scope of effect of a decision should be within the scope of control of the module containing that decision. This reduces intermodule communication problems. Structure charts are useful in modular designs since they explicitly show module connections and module parameters. Since they contain no decision blocks, decisions are deferred until the detailed design stage; this allows design to proceed at a fairly abstract level. Stevens, et al., characterize the technique as emphasizing simplicity, low coupling and high cohesion of modules, and predictability (i.e., a module operates the same every time it is called, independent of its external environment) [Ref. 29].

2. HIPO

HIPO (Hierarchy plus Input-Process-Output) is a graphical technique consisting of two basic components: a hierarchy chart and input-process-output charts. Even though it does contain textual material, like Structured Design, it relies heavily upon diagrams and, for this reason, is considered to be primarily graphical. Originally, HIPO was intended to be used as a documentation tool in conjunction with Structured Design. As an iterative, top-down process, HIPO first involves describing a function as a series of steps in terms of inputs and outputs. This is done via the input-process-output diagram which is developed concurrently with the hierarchy chart. At the completion of each level, those functions included in the process box are themselves then subjected to the same analysis. This iterative process permits abstraction of functions into modules which have the following properties:

1. single entry point
2. single exit point
3. small size
4. structured nature [Ref. 30].

Input-Process-Output Charts represent a good documentation technique for describing functions appearing in Hierarchy Charts; however, they seem to be deficient relative to Structured Design in their ability to communicate module interfaces, a critical issue in any reusable design methodology.

3. <u>DREAM</u>

DREAM (Design Realization, Evaluation and Modelling) is a nongraphical technique which utilizes a design description language called DREAM Design Notation (DDN). DREAM supports a hierarchical design approach in that DDN descriptions are of classes of components within a system which are decomposible into subcomponents in an iterative manner. An event based design method is used in which pertinent events in the system are identified, constraints upon these events are developed, system components necessary to produce these events are identified, and interactions among these components are developed. DREAM is supported by an automated program support system.

The ideological foundation for DREAM is the belief that architectural design requires a language which expresses specifications in requirements-oriented rather than implementation-oriented terms. Abstraction is therefore supported since specifications possess a behavior or effect orientation instead of an operation or cause orientation. This feature also facilitates definition of module boundaries and interactions. Riddle outlines the following attributes required of a design description language:

1. analysis oriented
2. unambiguous
3. modular
4. hierarchical

5. incremental
6. outward directed
7. redundant
8. modification oriented
9. guidance oriented
10. non-prescriptive [Ref. 31].

4. Jackson Design Method

Developed by M. Jackson, this method is a three step technique which involves both graphical and non-graphical notations. First, the structure of the data is graphically represented as trees; these data structures are then composed into a hierarchy of the program structure. Finally, the executable operations required are listed, and each is allocated to its proper place in the program structure. This list forms the basis for a program implementation since these operations are elementary executable statements of a program language. Every operation must have operands which are data objects [Ref. 32]. This methodology appears to emphasize the description of the problem and its solution, particularly as a step by step sequence of processes. This is in consonance with its classification as a detailed design technique; nevertheless, it hinders its ability to abstract and modularize functions.

5. Structured Flowcharts

Structured Flowcharting (also called Nassi-Shneiderman Charting) is a graphical technique developed as an alternative to traditional flowcharting in order to represent, visually, structured programming techniques. Its primary benefit is to document a design which has already been developed, and it is therefore useful during later stages of the design process. Claimed advantages include:

1. well defined and visible scope of iteration
2. well defined and visible scope of IF-THEN-ELSE
3. scope of local and global variables is obvious
4. arbitrary transfers of control are impossible
5. complete thought structures on one page [Ref. 33].

Nassi-Shneiderman diagrams represent standard programming constructs which might occur in any structured programming language. It should be noted that Structured Flowcharting is a low level technique, and as such, its ability to abstract is limited.

6. PDL

PDL (Program Design Language) is an algorithmic language which uses a semi-formal syntax and structured format to describe processes. Syntax and format can vary greatly from designer to designer. Like structured flowcharts, it is more appropriate during later stages of the design process, although its abstraction abilities do permit top-down, stepwise refinement from general concepts to specific implementations. Its primary purpose is to communicate ideas to people, not computers; however, there have been automated PDL's [Ref. 34].

IV. REUSABLE SOFTWARE DESIGN LITERATURE

In this chapter, the current literature on reusable software is reviewed to ascertain the current status of design vis-a-vis reusability. The literature on this subject is rather sparse, although the recent conference on reusable software sponsored by International Telephone and Telegraph in September 1983 indicates that this is changing. This review adopts a somewhat chronological approach, beginning with the oldest references first. Only points pertinent to the design phase are presented.

A. RATHEON EXPERIENCE

Although he was not the first to develop the idea of reusable software, Robert Lanergan can be credited as the first to popularize the concept in a series of articles beginning in 1978 on actual implementation of a reusable methodology at Raytheon, Inc. The importance of this work stems from his prolific writing on and personal advocacy of reusable software as a possible means of improving software development productivity. His work includes several design related issues which are outlined:

1. The methodology emphasizes functional modularity to develop reusable modules.
2. A top-down approach is used in which high-level logic is designed and coded first.

3. Programs are composed of two types of reusable modules (functional modules and logic structures) plus some unique factors.
4. Functional modules are designed and developed for a specific purpose. These are categorized and cataloged for reuse.
5. Logic structure is a cluster of small redundant functional modules designed hierarchically with a prewritten Identification Environment, Data and Procedure Divisions. There are three basic types of logic structures: (a) select and/or edit, (b) update and (c) report. Each of these may or may not have a sort. Crucial to the construction of a logic structure is a central supporting paragraph.

The Raytheon methodology is a COBOL-based methodology oriented toward business applications; indeed, the categorization of logic structures into the three basic types listed above is based on Lanergan's view that these are the "only...things you can do with a business computer program" [Ref. 35]. This methodology is interesting in that it advocates modularization of functions, top-down design, and a two level architecture composed of a framework logic structure to which interchangeable functional modules may be attached.

For all the credit which must be given him for popularizing the concept, there are, however, several criticisms of his work. First, his methodology does not seem to be generalizable to the field at large, although there is no indication that he intends it to be. As he says, the technique is ADP oriented. Nevertheless, a great service could have been performed if he had attempted to communicate his methodology better and to derive some general principles of reusable design. This points out a second criticism; his articles do not sufficiently explain his methodology. The reader is left with only a superficial grasp of his technique. A third criticism derives from the fact that his methodology is not an instance of "pure reusability"--components are reusable only within the same program systems[3]. A minor criticism concerns lack of substantiation of his reported productivity increases. Finally, Lanergan's articles and methodology emphasize reuse of code; as this thesis proposes, this is not where emphasis on reuse should be. For all these criticisms, his is still the most noted contribution to reusability to date.

B. ASSORTED REFERENCES

Between Robert Lanergan's initial paper in 1978 and the ITT Workshop on Reusability in 1983, there was only one paper devoted solely to the topic of reusability and software design, and that was Bertrand Meyer's paper. There are, however, a few brief discussions of the subject in various articles. Rauch-Hindin quotes a TRW employee's explanation for the

[3]Personal telephone conversation with Robert Lanergan.

non-acceptance of modules of reusable code as being in the assumptions made when these modules were originally designed. Reuse requires that the implementor change either his design or the module's code if these assumptions are not consistent with the assumptions of the current design. An implementor with many modules, each developed separately, faces complex interfacing problems and may be forced to alter his design to assure that everything fits [Ref. 36]. Rauch-Hindin quotes others as advocating that reusable modules must be specifically designed for reuse, must have simple interfaces, and must have well defined functionality or high cohesion (a single, well-defined purpose). The design must also incorporate the need for clean interfaces [Ref. 37].

In another article on reusable software, Rauch-Hindin again included some comments that are pertinent to reuse of design. First, she pointed out that a major obstacle to reuse of modules is lack of standardization of "meaning and representation of data and parameters." One of the reasons for the success of reusable mathematical subroutines has been their well-defined algorithmic specifications and parameteric information. She also discusses Ada "packages" which possess enhanced interface capabilities in their specification sections, as well as criticality of communicating information about reusable modules to potential users [Ref. 38]. Throughout her series of articles, Rauch-Hindin stressed several things regarding software reuse and design:

1. The importance of initially designing modules with reuse in mind; modules not designed in this way will be difficult to reuse.
2. The importance of the module user being able to acquire more information about reusable modules in addition to just the code itself, particularly assumptions made at design time.
3. The criticality of interface issues in linking modules.
4. The importance of decomposition into highly cohesive modules.
5. Standardization of "meaning and representation of data and parameters" as a means of enhancing reusability.

Bertrand Meyer addresses design of reusable software and proposes abstract data types as a possible solution to reuse in a 1982 paper. He sees the subprogram package--a group of routines called by any program--as being the key to implementation. These packages will be used by others if they are designed for simplicity, self-restraint, ease of use, homogeneity, and safety. By simplicity, he means that all information needed to use a package should be available in the package "without further reference to any written document." Self-restraint deals with the idea that subprogram packages perform general utility tasks and not esoteric or interesting problems. Ease of use includes good documentation and consistency of design (for example, standardized parameter order). By homogeneity, Meyer again is referring to subprogram parameters and need to limit their number, as well as naming conventions for subprogram names. Safety refers to error handling capabilities of the subprogram package. A subprogram package itself is "the conscious implementation of one or more abstract data types." In addition, this is supported by the concept of information hiding, which separates public and private information contained in these packages [Ref. 39].

C. JLC PANEL ON SOFTWARE REUSABILITY

As was mentioned in Chapter I, the Proceedings of the Joint Logistics Commanders Software Workshop held in 1981 containeda panel on software reusability; the panel specifically advocated reuse of intangible design as well as other products of the software life cycle. In a section on functional design reuse, the panel stated that reusable design must be "clearly identified, locatable, and understandable." It perceived modular partitioning as a particularly crucial step which requires unique and, as yet, undeveloped techniques. To be identifiable, a design must be described by the requirements it fulfills and functions it implements. The panel stated that a "top-down hierarchical exposition of the design is mandatory." A design should consist of several levels, each level showing successively more detail. This facilitates comprehensibility by limiting number of internal subprocesses necessary to be understood. The panel also advocated a standardized means of design exposition, including textual or graphical notation [Ref. 40].

D. ITT WORKSHOP ON REUSABILITY

The literature on software reusability has improved considerably with publication of the Workshop on Reusability in Programming in September 1983. This is not the case with respect to the topic of reusable design. However, there are several articles in which this issue is addressed, and each of these is analyzed separately.

1. Doberkat, Dubinsky and Schwartz

In their paper outlining their work with a very high level language, SETL, the above authors attempt a definition of reusability of design:

> ...the ability to convey the overall procedural structure and principal data designs of large complex programs to implementation groups wishing to develop the same or similar functions in new environments...

According to the authors, such an endeavor requires a specification language which allows the complete description of significant abstract features, while suppressing design-irrelevant details. Such a language does not now exist [Ref. 41].

2. Horowitz and Munson

Horowitz and Munson, in a general review of the current state of research on reusable software, devote a section to reusable design. They summarize the essential idea of reusable design as being a formal study of a particular application domain (domain analysis) and using "artifacts" of this study to develop an automated system. These "artifacts" include data types in the domain, and various constraints on these processes. Reuse occurs "when subsequent efforts to automate specific instances of the domain make use of the previous artifacts that were developed." A central problem lies in the medium by which these artifacts are recorded for reuse; specifically, in what language are they to be recorded, are they to be immediately translated into modules, or is there some other approach? The authors then describe two example approaches taken by Neighbors (DRACO) and Rice (Automatic Software Generation System) [Ref. 42].

3. Deutsch

Smalltalk-80, according to an article by Peter Deutsch, facilitates reuse of design through its data abstraction capabilities. Reuse of algorithms is hampered by their dependence on detailed implementation of the data they manipulate. Data abstraction circumvents this by packaging a data type with its parameters; when a procedure is invoked using objects of that data type, the correct parameters for that type are automatically substituted into the procedure. In Smalltalk-80 abstract data types are hierarchically arranged. Smalltalk-80 also employs the concept of a class, which consists of a description of a data object and a set of named procedures having access to this description.

Abstract classes can include reusable algorithms. Smalltalk-80 extends this idea through its concept of a framework, which is a collection of abstract classes and their algorithms. As Deutsch described it, "particular applications can insert their own specialized code (into the framework) by constructing concrete subclasses," thereby enabling reuse [Ref. 43].

E. SUMMARY AND ANALYSIS OF SOFTWARE DESIGN LITERATURE

Our primary impression of the current state of reusable software design as gained from a review of the literature to date on the subject is that it is very primitive. Obviously, the subject has not received a lot of attention. Most of the discussion is presented from a very narrow focus, usually based on observations derived from specific research conducted by the author. What is needed is a comprehensive analysis of the topic with a coherent, organized and comprehensive theoretical foundation for the subject. The synopsis presented below is the first step in doing this.

First and foremost, no single design approach emerges in the literature as preeminent; there are as many methodologies as there are authors. None of the traditional techniques, such as PDL, are advocated or even studied in the reusability literature. There does seem to be a consensus that any design methodology based on reusability must be a top-down approach. Most authors emphasize a software design technique which allows a high degree of functional modularization, abstraction of multiple levels in the design, structured design and programming concepts, and information hiding. More recently, the literature has begun to investigate domain analysis as a possible design methodology pertinent to the reusability problem. Aside from these areas of general consensus, there are other, less-agreed to ideas on the characteristics of a reusable design methodology:

1. In business ADP, reuable modules are of two types--functional modules and logic structures.
2. Reuse of design requires knowledge of assumptions under which the design was developed[4].
3. Interface issues are important at design time.
4. Software must be specifically designed for reuse in order to be reused.
5. Designs should strive for simple, clean interfaces.
6. Designs of modules should exhibit high cohesion.
7. Communication of the design to the user who will reuse it is important and is facilitated by certain characteristics of the design itself, such as simplicity, self-restraint, homogeneity, ease of use, and safety, as well as means of access to the design.
8. Standardized design notation/exposition is advocated.

---

[4]Professor Normal Lyons of the Naval Postgraduate School, Monterey, Ca., states that this is essentially what minicomputer original equipment manufacturers do when they specialize in a given industry. Much of the systems analysis work they do is reusable because they seldom venture outside that industry.

The above reinforces our initial comments on the haphazard state of current thinking; current literature is sparse, fragmented, and no overall authoritative work exists. Part of the problem lies in the way the reusability literature refers to design. There are actually two aspects to design; the first is design of the reusable components themselves; that is, what should they look like, while the second is design using reusable components; that is, what methodology should be used. The first aspect influences the second. This distinction should be kept in mind during the next chapter since it is the latter which is emphasized.

V. GUIDELINES FOR A REUSABLE DESIGN METHODOLOGY

This chapter seeks to develop guidelines for choosing or developing an appropriate design methodology which incorporates reusability. The approach is three-fold. First, a design scenario incorporating reuse of existing components is proposed; from this, those characteristics which a design methodology must possess are derived. Next, design methodologies reviewed in Chapter III are compared to see which possesses most of these characteristics. Finally, an idealized reusable design methodology is discussed.

A. A PROPOSED REUSABLE DESIGN SCENARIO

To date, no one has proposed a concrete methodology of software design which incorporates reuse of components. Lanergan describes his overall approach in general terms without really honing in on design specifics, while Rice and Schwetman compare reusable software development to an eight step electronic component fabrication process [Ref. 44]. Chandersekaran and Perriens briefly mention a three stage approach to building new software systems from existing parts [Ref. 45]. This situation is remiss; before an appropriate technique can be chosen or developed, one must know what the reusable design process will look like. For this reason, the following scenario is proposed.

A designer starts from a set of requirements, either formal or informal. From this, he develops an appropriate structure for the overall design. He then begins a process of design decomposition into smaller design modules, and for each module, he checks to see if an existing design component exists which can be plugged into the overall design at this point. If it does, then that portion in the design is complete, and he proceeds to other modules. If not, decomposition of that module continues. When decomposition is no longer possible and a reusable module for a particular function does not exist, only then is a new module designed, using a standardized methodology which emphasizes designing modules for reuse. Also, if it is possible to reuse an existing component by modifying it, this is permissible, but a completely different component is thus created.

When flaws in the design are discovered, correction involves isolating the appropriate module or modules' location(s) and redesigning via the above process. Isolation of location means determining which levels in the decomposition are affected.

Code for modules designed in this manner are produced in one of two ways; first, each module is coded at completion of detailed design and the code archived along with design notation, design assumptions and documentation. Alternatively, only these latter three are archived, and coding waits until an actual implementation is required. This latter method seems preferable in terms of storage cost and implementation language flexibility. Thus, the designer's job is finished when he has assembled a complete, detailed design composed of both previously developed and new modules in the design notation. At this point implementation into code is an automatic process. Reusable code, as currently envisioned, would not exist under this option. For each particular existing component, there could be standard sets of source code statements in several possible program languages.

From the above, it is evident that concern should not be with structure or appearance of coded modules; instead, it is with appearance of design level modules, both architectural and detailed design levels. It is envisioned that a designer at any stage in the architectural or detailed design process may find a previously built module which appropriately fulfills a function within the overall design. This module may itself be composed of many previously built modules. At the lowest level of detail design, each module is a small, highly cohesive module; this smallness implies a large number of modules, emphasizing what has consistently been pointed out in the literature; i.e., interfaces are extremely crucial.

No one at this point in time can say what reusable software will ultimately look like; for now, the best estimate is that reusable software will be small packages. These packages will be highly modular, performing a single, well-defined function. The packages will consist of the architectural design notation, design assumptions, detailed design notation, and documentation. Ideally, the detailed design notation will be standardized and of a generic nature, in order to permit implementation in any programming language. It should be of a form so that source code can be easily produced, directly from the detailed design notation, once appropriate program language is selected.

B. REUSABLE DESIGN METHODOLOGY CHARACTERISTICS

From the scenario proposed in the last section, the characteristics of a design methodology which would facilitate reuse of components can be inferred. Such a methodology should:

1. facilitate modular decomposition
2. possess a *top-down* approach
3. permit abstraction of lower levels during decomposition (although these first three are closely related, there is not necessarily a one for one correspondence; i.e., a methodology may greatly facilitate a top-down approach to design but be less effective in enhancing abstraction).
4. be composed of one methodology which incorporates facilities for both architectural and detailed design
5. possess a textual design notation at the architectural level, possibly supplemented by a graphical notation
6. possess a textual notation at the detail design level

7. provide for a correspondence between modules at the architectural level and modules at the detail level (i.e., if a particular architectural module is selected, a certain corresponding detail design module or modules would follow)
8. permit textual commenting, separate from architectural and detailed design notations, to accommodate the need for assumptions and documentation.

From the previous chapter, the literature indicates that this methodology should also:

9. support information hiding (specifically, it should not be necessary to know the implementation of one module in order to implement another module)
10. support simple, clean module interfaces and explicit parameterization (the ability to define number, form and type of parameters and to do so in a manner which is not cumbersome to the prospective user is a very important element of any reusable methodology)
11. be simple and easy to use
12. possess a standardized format known to all users.

C. CURRENT DESIGN METHODOLOGIES COMPARISON

The task of comparing existing design methodologies to see which best facilitates a reusable approach can most easily be accomplished via Table 1. Along the vertical axis, each of the twelve factors cited in the previous section is listed; horizontally, each of the six existing design

TABLE 1

CURRENT DESIGN METHODOLOGIES COMPARISON

	Structured Design	HIPO	DREAM	Jackson Design	Structured Flowchart	PDL	Max = 18
Modular Decomposition	3	3	3	1	0	3	13
Top Down	3	3	3	1	1	3	14
Abstraction	3	3	3	2	1	3	15
One Methodology	0	0	0	0	0	1	1
Textual Architectural Notation	1	1	3	0	0	1	6
Textual Detailed Notation	0	0	0	1	0	3	4
Architectural-Detailed Correspondence	0	0	0	0	0	0	0
Textual Commenting	0	0	0	0	0	1	1
Information Hiding	3	3	3	1	0	1	11
Clean Inter-Faces/Explicit Parameters	3	1	3	0	0	0	7
Simple, Easy to Use	2	3	1	1	3	3	13
Standard Format/ Language	3	3	3	1	3	1	14
Max = 36	21	20	22	8	8	20	

0 = None 1 = Low 2 = Medium 3 = High

techniques mentioned in the previous chapter is listed. For each design technique, a number is placed in the appropriate row indicating extent to which that technique facilitates that particular factor listed on the vertical axis. The numbers range from zero (does not facilitate) to three (strongly facilitates); thus for example, HIPO is rated high in abstraction capabilities (3), while it is rated as having no textual commenting capabilities (0). In addition, each column is summed to give a relative ranking of each particular design technique in relation to the others and in relation to an ideal maximum score of 36, this being the best score attainable if a methodology possessed all characteristics to the maximum extent required for implementing reusable design. Finally, each row is summed to give an indication of the current status of existing design methodologies as regards their ability to support each of the twelve requirements of a reusable design methodology. Maximum score possible in this case is 18.

As Table 1 shows, no existing design methodology completely meets all the criteria required to support reusable design. Of 36 possible points, DREAM scores highest of any methodology with 22; Jackson Design Methodology and Structured Flowcharting score least with eight each. A significant but expected pattern also occurs in that, in general, architectural design methodologies' scores seem to be decidedly higher than those for detailed design techniques, PDL being the exception. When the row scores for each of the twelve categories are compared to the maximum of 18, wide variation is encountered. Current methodologies are most inadequate in the category "Architectural-Detailed Correspondence" (score = 0), while highest in the "Abstraction" category (score = 15). The following categories are considered very deficient (scores less than 50% of maximum possible):

1. One Methodology
2. Textual Architectural Notation
3. Textual Detailed Notation
4. Architectural-Detailed Notation Correspondence
5. Textual Commenting
6. Clean Interfaces/Explicit Parameters.

It is concluded from this that no existing design methodology is adequate for reusable design and that this is probably one of the main reasons why reusable software has not been more widely implemented.

D. AN IDEALIZED REUSABLE DESIGN METHODOLOGY

Ideally, a reusable design methodology supports all twelve characteristics mentioned in the previous section. Such a methodology allows the designer more freedom at higher levels in the design to decompose as he sees fit, while providing opportunities at lower levels to incorporate previously designed modules. At the lowest level of detail, there is a direct correspondence between textual design statements and source code implementation. Linkage between architectural textual statements (and graphical notation) and detailed text is also direct. The methodology possesses a standardized, formal syntax and grammar.

The design notation must possess capabilities to communicate module interfaces effectively; this is probably the most important feature of the design language. The language must allow accurate representation of required module inputs and generated outputs, including their number, types, and order [Ref. 46]. We suspect that the more "generic" a module is, the larger and more complex it will be; this results in more complicated interfaces. This is at odds with Meyer's advocacy of short parameter lists, and it argues for larger numbers of "non-generic" modules.

VI. OTHER ISSUES

There are a number of issues important to the topic of reusable software, and, as such, they relate to the main topic of this thesis, reusable software design. These issues are primarily of an administrative nature, such as that of project organization. In discussing these issues, the format used by the panel on reusability of the Joint Logistics Commanders Software Workshop is adopted [Ref. 47]; this format addresses all major administrative issues in reusability. During this discussion, an attempt is made to emphasize those aspects which are of importance to the central theme of design. As comments about reusable software are made, the reader should bear in mind that they apply directly to reusable design.

A. PROJECT ORGANIZATION AND CONTROL

It is likely that the actual adoption of reusable methodologies will be done by individuals and organizations who perceive them as being more cost effective, easier to use, and/or more time-saving than traditional software development methodologies. This thesis agrees with the JLC panel's position that adoption of reusable development methodologies will require differences in or special emphasis in certain areas of project management. JLC feels that these differences justify creation, within each organization developing software, of an activity devoted to overseeing reusable software, the Reusable Software Organization (RUSO). This author does not feel at this time that establishment of such a specific organizational entity to oversee reusability is required. Actual implementations of reusable software at Raytheon [Ref. 48] and Hartford Insurance [Ref. 49] have not been accompanied by establishment of RUSO-like activities.

There is need, however, for assurance that functions which a RUSO might be responsible for are carried out. These would include:

1. establishment and monitoring of organizational reuse goals
2. development of strategic and tactical reuse plans
3. establishment of reuse standards and guidelines
4. evaluation of reuse methodologies and tools
5. responsibility for reuse quality assurance
6. establishment and maintenance of reuse archival facilities and activities.

Actual establishment of a RUSO is best dictated by size of the organization and software development effort.

B. QUALITY ASSURANCE/TESTING

In the areas of quality assurance and software testing, there exist two distinct views regarding reusable software. The first says that because this software is reused among many different programs in a variety of environments, it must be subjected to a rigorous testing and certification procedure. In essence, a higher level of testing than on non-reusable software is imposed initially. However, once a piece of reusable software is thus certified, it may be used by a wide range of users with confidence and with reduced need for repeated testing in every new application into which the reusable component is inserted. The second view derives from the philosophy that since testing only reveals presence of errors and never proves their total absence, development of perfect software through testing is impractical. This view says that a more practical approach to reusable software certification, and one which will encourage production of reusable components, is merely to require the component's developer to document adequately the level of testing to which a component has been subjected. It is then the responsibility of the user of that component to test beyond this level if necessary. This last point highlights a problem with the first view, in that not all applications require the same standards of testing--some applications are less critical. Why require such strict certification standards when they are not necessary in all cases? Finally, reduced testing standards may encourage component development.

At the intra-organizational level, there probably is need for some entity to establish minimal quality assurance/testing/certification standards. As the JLC report recommends, this appropriately belongs to a RUSO-like activity.

Also the JLC report stated that one important result of utilizing previously tested components is an overall reduced level of testing at the component level, although this statement should be tempered by the analysis in the previous paragraph. System integration testing would not be affected. In addition, the JLC report recommends a top-down approach to testing, although no rationale for its preference over other testing methods is given. For this and other reasons, appropriate testing methodologies for reusable software is an excellent subject for further research.

C. REUSE ARCHIVAL ISSUES

Aside from development of an actual reusable software methodology, development of systems and techniques for archiving reusable software is the most important task to be accomplished. Importance of this area to an effective reusable methodology cannot be overemphasized, for it is ease of locating required components which will make or break the methodology. Additionally, this thesis feels that to the greatest extent possible, design of the archival system should depend upon design of the reusable methodology; that is, we think that designing archival systems before concrete details of the methodology are developed is like "putting the cart before the horse."

It is possible, based on analysis in this thesis, to make some predictions regarding reusable archival systems. First, if the organization establishes a RUSO, then that body should establish a "library" to perform this archival function whose primary task is to address recapture of software components. It is uncertain at this time what form the archival entity will take, whether or not a "library-like" form is best remains to be seen. JLC also made an excellent recommendation on establishment of a Reusable Software Change Control Board (RCCB) whose purpose is "to maintain the integrity of the (reusable software) and its library." It is "sole authority for the approval, disapproval, or deferral of proposed changes." Someone within the RUSO should approve all components entering the library. Primary activities of the library would involve distribution and dissemination of reusable software.

The library itself would probably consist of a combination of manual and automated archival facilities. Foremost of these are indexing and cataloging facilities for storage and retrieval of reusable software developed in house, as well as facilities for indexing externally developed reusable software. These latter might be similar to components catalogs.

From the analysis in previous chapters, it is felt that, from a design standpoint, library facilities should be oriented toward archival of textual material, although capabilities should be present to handle graphical notation as well. However, when the entire range of material developed during the entire reusable software life cycle is considered, the orientation might be different. As to what sorts of things will be archived, the following are postulated:

1. Architectural Design Notation
2. Detailed Design Notation
3. Design Assumptions
4. Documentation
5. Test Plans and Results
6. Code.

As was discussed in an earlier chapter, the first four items above exist as modules designed in a standardized reusable design language; archival considerations should take this into account. Test plans would not be in this same language, but facilities for linking a module's design language, test plans and code should exist. Presence of code in the library depends on the overall reusable methodology itself.

There are numerous other issues involved in reusable software--reimbursement, program language selection, reusable software maintenance, and so on. This chapter attempts to address only the most important. Certainly, further research on those discussed, and those not discussed, is warranted.

VII. CONCLUSIONS AND RECOMMENDATIONS

In studying reusable software, its most outstanding characteristic is its newness as a topic of research and analysis. Although reuse of software has existed for some time, consideration of the idea does not appear in the literature until the late 1970's. It is only within the last two years that reusable software has received erious analysis. There are two basic reasons why this topic is generating such increased enthusiasm; first is that it is seen as a possible solution to the software crisis. All trends show an explosive growth in demand for software products, and reuse of these products after development represents a common-sense solution. Second, the idea that software is a capital asset is a novel way of looking at something that should have been apparent from the start. In the purest definition of the term, a capital asset is something which is not "used up" during its operation; this is a primary characteristic of software. Interest in reuse of software stems from an unconscious realization that it possesses this characteristic of all capital assets.

As with all new endeavors of study, reusable software is undergoing a definitional phase. There are several aspects to this statement. First, there is no commonly accepted, authoritative definition of the term; it is strictly a "buzzword." Hopefully, the attempt in Chapter II can serve as a beginning for dialog in this area. Second, there is no generally accepted organization to or boundaries for research--there is no "taxonomy" of the subject. Therefore, the literature contains almost as many directions of research on reusable software as it does authors. This thesis proposes the software life cycle as the most appropriate structure for organizing thought and research on this topic. Finally, lack of an accepted definition results in a situation where everyone has his own idea as to what is reusable. This thesis advocates reuse of architectural design, detailed design, design assumptions, code, documentation and test plans--essentially, all the products of the life cycle.

The particular emphasis of this thesis, software design, is chosen because it is felt to be the most crucial product of the life cycle from a reuse standpoint. The reviews of existing software design methodologies included in Chapter III and of literature on reusable software design in Chapter IV show how no current design methodology is suitable from a reuse standpoint and how woefully inadequate is research into this area. Accordingly, a reusable design scenario is developed in Chapter V as a basis for developing guidelines for a design methodology which incorporates reuse of existing components. Development of this methodology is seen as the most important step to be taken in realizing reusable software.

Chapter VI is an attempt to overview briefly several other topics of importance to reusable software in general, and reusable design in particular. However, decisions on these areas are deemed appropriate only after decisions regarding design methodology have been made. This does not minimize their importance to the effort.

Recommendations of this thesis are centered on the strong need for further research into topics which we have only been able to explore briefly. First and foremost, a comprehensive, authoritative review of the entire subject of reusable software is needed; this task is much too large for a thesis-length endeavor. Second, further research into the specific area of reusable design methodologies is needed. Two questions which might be pursued are: (1) Can present methodologies be adapted to accommodate reuse, and (2) What characteristics should such a methodology possess? Research into the other issues mentioned in Chapter VI is also required; the areas of archival requirements and quality assurance requirements present excellent future thesis topics.

LIST OF REFERENCES

1. Dijkstra, Edsger, "Go To Statement Considered Harmful," Communications of the ACM, Vol. 11, No. 3, pp. 147-48, March 1968.

2. Parnas, D.L., "On the Criteria To Be Used in Decomposing Systems into Modules," in Tutorial on Software Design Techniques, 3rd ed., ed. Peter Freeman and Anthony I. Wasserman, p. 221, IEEE, 1980.

3. Ibid.

4. Standish, Thomas A., "Software Reuse," Workshop on Reusability in Programming, ITT, p. 45, September 1983.

5. Horowitz, Ellis, and Munson, John E., "An Expansive View of Reusable Software," Workshop on Reusability in Programming, ITT, P. 254.

6. Lanergan, R. and Poynton, B., "Software Engineering with Standard Assemblies," Association for Computing Machinery Proceedings of the Annual Conference, Washington, D.C., pp. 507-14, December 1978.

7. Lanergan, R. and Dugan, Denis K., "Software Engineering with Reusable Designs and Code," Association for Computing Machinery Proceedings of the Annual Conference, pp. 296-303, Fall 1981.

8. Lanergan, R. and Grasso, Charles A., "Software Engineering with Reusable Designs and Code," Workshop on Reusability in Programming, ITT, pp. 224-27.

9. Rauch-Hindin, Wendy, "Software Tools: New Ways to Chip Software into Shape," Data Communications, Vol. 11, pp. 83-8, April 1982.

10. Rauch-Hindin, Wendy, "Reusable Software," Electronic Design, Vol. 31, No. 3, pp. 176-94, 3 February 1983.

11. Wegner, Peter, "Reflections on Capital-Intensive Software Technology," Software Engineering Notes, Vol. 7, pp. 24-33, October 1982.

12. Wegner, Peter, "Varieties of Reusability," Workshop on Reusability in Programming, ITT, pp. 30-44.

13. Freeman, Peter, "Reusable Software Engineering: Concepts and Research Directions," <u>Tutorial on Software Design Techniques</u>, 4th ed., pp. 63-76, 1983.

14. Joint Logistics Commanders Joint Policy Coordinating Group on Computer Resource Management, "Report of the Panel on Software Reusability," <u>Proceedings of the Software Workshop</u>, 1 November 1981.

15. <u>Workshop on Reusability in Programming</u>, ITT.

16. Horowitz and Munson, p. 253.

17. Standish, p. 46.

18. Wegner, Peter, "Reflections on Capital-Intensive Software Technology," p. 24.

19. Ibid, pp. 24-25.

20. Freeman, p. 64.

21. Joint Logistics Commanders Joint Policy Coordinating Group on Computer Resource Management, p. 3.

22. Freeman, p. 63.

23. Freeman, Peter, "The Nature of Design," <u>Tutorial on Software Design Techniques</u>, 3rd ed., p. 47.

24. Ibid, pp. 46-7.

25. Ibid, p. 47.

26. Wasserman, Anthony I., "Information System Design Methodology," <u>Tutorial on Software Design Techniques</u>, 3rd ed., pp. 30-1.

27. Freeman, Peter, "The Nature of Design," pp. 48-9.

28. Stevens, W.P., Myers, G.J. and Constantine, L.L., "Structured Design," <u>Tutorial on Software Design Techniques</u>, 3rd ed., p. 244.

29. Ibid., pp. 244-52.

30. Stay, J.F., "HIPO and Integrated Program Design," <u>Tutorial on Software Design Techniques</u>, 3rd ed., pp. 253-7.

31. Riddle, W.E., "An Event-Based Design Methodology Supported by DREAM," <u>Tutorial on Software Design Techniques</u>, 3rd ed., pp. 269-81.

32. Jackson, Michael A., "Constructive Methods of Program Design," _Tutorial on Software Design Techniques_, 3rd ed., pp. 395-6.

33. Yoder, Cornelia M. and Schrag, Marilyn L., "Nassi-Shneiderman Charts: An Alternative to Flowcharts for Design," _Tutorial on Software Design Techniques_, 3rd ed., pp. 386-93.

34. Caine, Stephen H. and Gordon, E. Kent, "PDL--A Tool for Software Design," _Tutorial on Software Design Techniques_, 3rd ed., p. 380.

35. Lanergan and Poynton, p. 508.

36. Rauch-Hindin, Wendy, "Software Tools: New Ways to Chip Software into Shape," p. 104.

37. Ibid., p. 107.

38. Rauch-Hindin, Wendy, "Reusable Software," pp. 176-94.

39. Meyer, Bertrand, "Principles of Package Design," _Communications of the ACM_, Vol. 25, No. 7, pp. 419-28, July 1982.

40. Joint Logistics Commanders Joint Policy Coordinating Group on Computer Resource Management, pp. 5-10.

41. Doberkat, Ernst, Dubinsky, Ed and Schwartz, J.T., "Reusability of Design for Complex Programs: An Experiment with the SETL Optimizer," _Workshop on Reusability in Programming_, ITT, pp. 106-8.

42. Horowitz, Ellis and Munson, John E., "An Expansive View of Reusable Software," _Workshop on Reusability in Programming_, ITT, pp. 250-62.

43. Deutsch, Peter L., "Reusability in the Smalltalk-80 Programming System," _Workshop on Reusability in Programming_, ITT, pp. 72-6.

44. Rice, John R. and Schwetman, Herbert D., "Interface Issues in a Software Parts Technology," _Workshop on Reusability in Programming_, ITT, p. 130.

45. Chandersekaran, C.S. and Perriens, M.P., "Towards an Assessment of Software Reusability," _Workshop on Reusability in Programming_, ITT, p. 180.

46. Meyer, pp. 421-2.

47. Joint Logistics Commanders Joint Policy Coordinating Group on Computer Resource Management, pp. 19-25.

48. Lanergan and Grasso, pp. 224-7.

49. Cavaliere, Michael J. and Archambeault, Philip J., Reusable Code at the Hartford Insurance Group," Workshop on Reusability in Programming, ITT, pp. 273-78.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, VA 22314	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93943	2
3.	Prof. Gordon Bradley, Code 52Bz Department of Computer Science Naval Postgraduate School Monterey, California 93943	4
4.	Prof. Norman Lyons, Code 54Lb Department of Administrative Sciences Naval Postgraduate School Monterey, California 93943	1
5.	Raytheon Company Hartwell Road (BLA1-3) Attn: Robert Lanergan Bedford, Mass 07130	1
6.	Commanding Officer Naval Medical Data Services Center Bethesda, MD 20814	2
7.	LCDR William C. Johnson MSC USN 3033 Morning Trail San Antonio, TX 78247	2
8.	Computer Technologies Curricular Office Code 37 Naval Postgraduate School Monterey, California 93943	1